E-BOOK

TEACHING AN ELEPHANT TO DANCE

Intentional evolution across teams, processes, and applications

Burr Sutter, *director of developer experience*
Deon Ballard, *content marketing*

WHAT DOES CHANGE MEAN?
NOTICING THE ELEPHANT IN THE ROOM
A DARWINIAN VIEW OF DIGITAL TRANSFORMATION
"YOU CAN'T HAVE CONTINUOUS DELIVERY:" CONWAY'S LAW, DEVOPS, AND CULTURE
- Culture first
- DevOps as a first step

DESIGNING AN APPLICATION ARCHITECTURE: LOOKING AT MICROSERVICES
- Design stamina, technical debt, and strategy
- Microservices and monoliths, defined
- Fallacies of distributed computing
- Rethinking a meaningful application

DRIVING FAST (RESPONSIBLY)
- Self-service, automation, and CI/CD
- Advanced deployments and innovation

HOW TO TEACH AN ELEPHANT TO DANCE
- Choose your stage
- Define your operating principles
- Strangle your monolith

CONCLUSION

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com

Applications have moved outside the IT department. There's a truism that all companies are now software companies, and the ability to rapidly provide new services and new functionality to customers is one of the key competitive differentiators a company can offer. IT agility is a stone that startup Davids can use to unseat massive Goliaths.

Once upon a time, several generations ago (in technology years), IT departments were internal departments, focused on maintaining infrastructure and services within the company. Some companies may have had external-facing services, particularly web services, but this was still generally a narrow and restricted area. IT wasn't a revenue-generating or strategic department; it was a supporting environment viewed as a cost center.

One of the outcomes of an infrastructure-focused environment is that developers lost a sense of what their code was doing. Release cycles were long, and changes were slow. A developer would work on something and throw the code into testing or operations, and it would be released months later. Because of that long lead time, engineers lost the joy of being a developer—of creating something and seeing it work in real life.

One of the great, powerful changes with digital transformation and related cultural and technology changes like DevOps is that it reintroduces the joy of creating code. Developers can create something and then actually see it run. That's a powerful shift. It brings back the immediacy of creating code. Seeing application live provides developers with a feedback loop so that they can redesign and improve their code and make their projects thrive.

The elephant is where your organization is today. There are phases of change between traditional environments and modern environments powered by microservices and DevOps. Some organizations have the luxury to start from scratch, but for a lot of businesses, the challenge is teaching their lumbering elephant to dance like a nimble ballerina.

WHAT DOES CHANGE MEAN?

Digital transformation is a strategic change for businesses. It allows companies to pivot their core services as competitive pressures change, or as new regulations emerge, and to roll out updates as soon as a vulnerability is discovered.

However, there is no common definition of digital transformation aside from "changing things." The term digital transformation is sometimes used to mean new architectures, like microservices, or new processes, like DevOps, or new technologies, like containers and application

programming interfaces (APIs). When something can mean anything, it effectively means nothing. Digital transformation isn't a specific thing that you can get. It is a thing that every organization has to define uniquely for itself.

There is no single architectural pattern or technology platform that works flawlessly in every single environment. The organizations that win at digital transformation are the ones that have the clearest understanding of their own goals and culture, and it looks different for each of them. For example:

- Walmart deployed code on Black Friday while 200 million people were online.[1]
- Amazon deploys code updates every second (50 million per year) across hundreds of applications and millions of cloud instances.[2]
- Etsy does 60 deployments per day with a monolithic application.[3]
- Netflix deploys hundreds of time a day on a complex distributed architecture, with a single code change going from check-in to production in 16 minutes.[4]

Each of those enterprises is working with wildly different team structures, underlying technologies, code bases, and architectures. The point isn't that they focused on this one pattern or this one technology and everything worked. They all started with assessments of their teams, their current technical debt, and their business strategies—and then intentionally and coherently moved their enterprises in the direction they wanted to go. And then they got results.

That is the teaching process for getting an elephant to dance. Wherever your enterprise is now, you can move it to where it needs to be (technical debt, design flaws, and all)—with intentionality and a clear strategy. Meaning you have to have a clear understanding of what you are trying to accomplish and how far that is from where you are today.

NOTICING THE ELEPHANT IN THE ROOM

Assessing your current technical landscape and refining a business strategy are hardly simple tasks. It's difficult to get that kind of perspective. There is a well-known parable of six blind men who encounter an elephant, and each feels a different part, trying to identify the new beast. One feels the trunk and thinks it's a spear; another feels the side and thinks it's a wall; another feels the ears and thinks it's a fan. What's interesting is that the parable has several different endings, depending on the source. In some, the men are unable to accept the others' statements and they have a falling out. In another, a sighted man comes along and explains the overall appearance, unifying their perceptions.

The different endings are probably the most realistic part of the parable. The point of the story is that everyone has different perspectives, limited sets of information, and assumptions based on that perspective. The outcome of those different perspectives speaks to the inherent communication and

1 *O'Maidin, Cian. "Why Node.js Is Becoming The Go-To Technology In The Enterprise." NearForm, 10 Mar. 2014, www.nearform.com/blog/node-js-becoming-go-technology-enterprise/. Accessed 1 Sept. 2017.*

2 *McKendrick, Joe. "How Amazon Handles a New Software Deployment Every Second." ZDNet, 24 Mar. 2015, www.zdnet.com/article/how-amazon-handles-a-new-software-deployment-every-second/.*

3 *"The Great Microservices Vs Monolithic Apps Twitter Melee." High Scalability, 28 July 2014, http://highscalability.com/blog/2014/7/28/the-great-microservices-vs-monolithic-apps-twitter-melee.html.*

4 *Bukoski, Ed, et al. "How We Build Code at Netflix." The Netflix Tech Blog, 9 Mar. 2016, https://medium.com/netflix-techblog/how-we-build-code-at-netflix-c5d9bd727f15.*

relationships within the team—some can reach outside their perspective, some just fall apart. What we see depends on who we are, where we are, what we're looking for, what we know, and what we *don't* know.

This is further complicated by the fact that most existing organizations have more than one elephant. Netflix is a unicorn, not just because of its advanced microservices architecture and container-based testing and deployment environments. It had the ability to build its team processes and technology stack based on its business strategy, without technical debt. It was a startup.

Any organization with legacy applications (and legacy teams) has more than one elephant in its room, including:

- Current team structures and communication patterns.
- Development, testing, build, and release processes.
- Technical debt and existing commodity applications.
- Differing goals or strategic visions between departments.

Those are all elephants, and each stakeholder can potentially view those elephants from different perspectives.

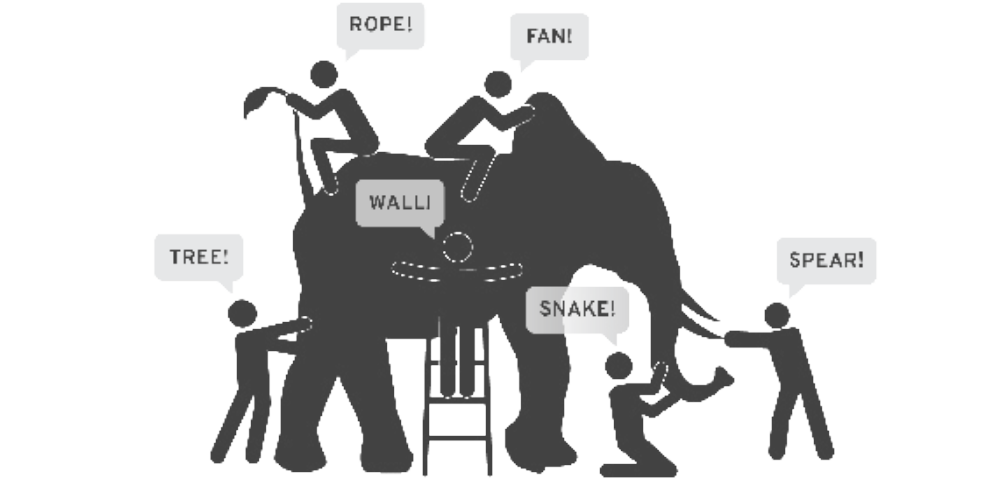

Figure 1. The allegory of six blind men and an elephant

A DARWINIAN VIEW OF DIGITAL TRANSFORMATION

There is a tendency to look at IT or digital transformation initiatives as binary: you do one thing or you do this other, different, thing. That may not be effective for a couple of reasons: First, because sometimes you can choose to do more than one thing or do a hybrid solution. Second, because frequently it's not a matter of changing one factor—different types of change may require different foundational changes or may depend on cultural and process changes.

Nothing happens in a vacuum.

It can be more constructive to look at digital transformation as a continuum, with different phases along the way that enable the next stage of evolution.

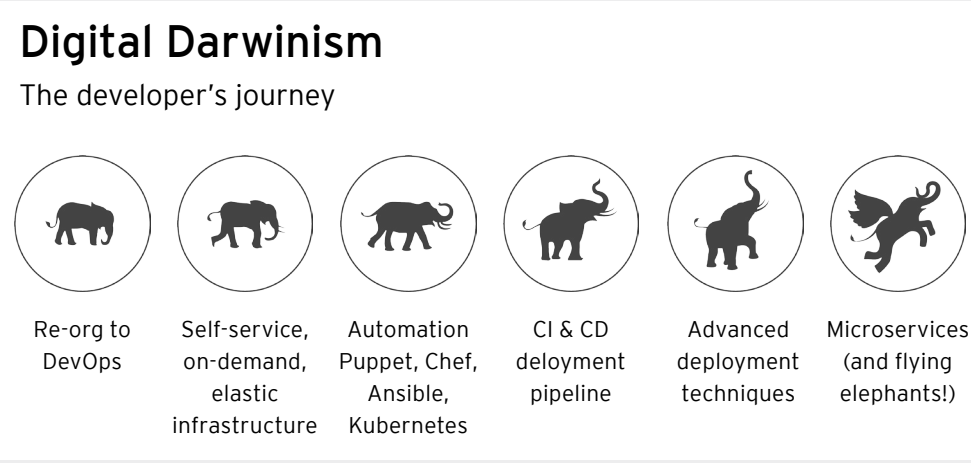

Figure 2. The stages of digital Darwinism

DevOps and process change

The foundation of digital evolution is DevOps. Applications, as with business strategy, are a reflection of the teams and communication that went into its creation. DevOps—or similar process changes—pulls more stakeholders into development discussions and offers broader insights into how operations maintains software and infrastructure (and how customers and partners are actually using those applications). It creates a tighter feedback loop between teams and requires open lines of communication. This open communication is the foundation for any other stage of evolution.

Self-service infrastructure

This phase is a technology-centric change, introducing efficiencies that are typically associated with modern technology platforms. Containers and self-service catalogs allow developers, testing, and operations groups to spin up consistent environments very quickly—in some organizations, taking the lead time for new instances down from days to minutes. Why should a technologist wait days for a computing resource?

ARE MICROSERVICES A REQUIRED STATE?

The final stage of digital evolution is microservices because of the complexity required to maintain it. But does that have to be the final stage of evolution for your organization?

Not necessarily.

If the other aspects of evolution are in place, a monolithic architecture can still release weekly and use advanced deployment techniques, CI/CD, and distributed, scalable infrastructure.

Microservices should only be considered when team sizes are large and you need to release faster than every week or on different schedules. A small team or small codebase does not need to be carved up into a microservices codebase.

Build automation and orchestration

Build automation is a two-part change. There is a technology angle, with advanced deployment engines like Red Hat® Ansible or Puppet, but it also requires a process change. Many organizations have strict processes in place around change and risk management; without adapting those processes into more agile methodologies, it would not be possible to take advantage of the new technology.

Continuous integration / continuous delivery (CI/CD) pipelines

Continuous delivery is a commitment to make delivery changes to software rapidly and iteratively. The idea of a pipeline is that there are both processes and technology in place that reduce the risk of poor quality (or broken) code from making it to deployment. This level shows the maturity of the previous steps—DevOps and open communication between teams, processes in place around testing and builds, and automated testing and deployment. When all of those stages are solid, then it is possible to push code through quickly. That's your pipeline.

Advanced deployment paths

Once the processes and infrastructure is in place for rapid deployments, then it is possible to start using your deployment systems as a way to mitigate any risk from updates, evaluate the effectiveness of functionality, and provide a real-life testing ground for new ideas. This can include having separate environments and load-balancing between them during deployments (blue-green deployments), using two different environments to test user interaction (A/B testing), or rolling out updates to small percentages of users and increasing that number safely (canary deployments).

Microservices (or distributed systems)

A microservice is a small application that performs a discrete, singular function. The overall application architecture may need to perform dozens or hundreds of different functions, and each of those functions would be defined and orchestrated in a microservice. A microservices architecture (or any distributed computing architecture) is simultaneously complex and simple. Individual services are much simpler and easier to maintain, add, and retire, yet the overall architecture has more complexity. When done right, a "microservice-based design is the ultimate manifestation of everything you learned about good application design."[5] This highly distributed architecture allows easier paths to scale, makes it easier to introduce new services or updates, and reduces the risk of a system-wide failure. That elasticity within the architecture is why microservices are widely associated with disruptively innovative companies like Netflix and Google.

"YOU CAN'T HAVE CONTINUOUS DELIVERY:" CONWAY'S LAW, DEVOPS, AND CULTURE

Rachel Laycock, at a DevNation 2016 general session, said that "'operationalizing' software is very hard."[6] She was sharing a story about a failure to implement continuous delivery at a gigantic financial services firm, which was suffering from taking weeks to roll out updates (even critical updates). The firm thought the fix was to move to continuous delivery. After six months of trying to change their processes, she finally went back and told one of their vice presidents that, no, they couldn't have continuous delivery.

5 *Cotton, Ben. "From Monolith to Microservices." 3 Jan. 2017, https://www.nextplatform.com/2017/01/03/from-monolith-to-microservices/.*

6 *Laycock, Rachel. "Continuous Delivery." Afternoon session. Red Hat Summit - DevNation 2016, 1 July 2016, San Francisco, California. https://www.youtube.com/watch?v=y87SUSOfgTY*

"Delivering fast and often is CI/CD. Supporting what you deliver is DevOps."[9]

– BURR SUTTER

Part of the problem is technology and architecture. The financial services company had a codebase with over 70 million lines of code and the apocryphal ball of mud architecture. But the software was the easy part; it was the obvious problem. The real elephant in the room was the organization's inability to change the behaviors of its various departments. That blocked its efforts to change.

Culture first

One critical thing to notice in the path of Darwinian (software) evolution is that it is not only a change of technology. It alternates between process and people changes and infrastructure changes, and the culture changes are infinitely more important. As Laycock concluded in her session address:[7]

"You can create the most beautiful architecture diagram you want. Once you involve these things—people, processes—you have to create the [cultural] environment that supports continuous delivery and the architectural discipline, because a process or structure change doesn't really have lasting change. People don't follow rules. So the real elephant in the room is Conway's Law. What does that mean? Legacy organizational structures will destroy your beautiful architecture *every. single. time*."

The key to looking at software or technology as an evolutionary change is that evolution is a natural function of its environment. In a business, that's the culture. The changes necessary to support evolutionary change can be supported by management, but they can't be dictated by management. People need to want to change. It's a matter of free will, not force.

Gartner actually has numbers for this: "90% of organizations attempting to use DevOps without specifically addressing their cultural foundations will fail."[8]

Changing the infrastructure or the application architecture is easy. To effectively change what you produce, you need to change your culture first.

Conway's Law states "any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure." This has two related interpretations:

- Changing your architecture or infrastructure will not change anything unless you also change your communication structure.
- Changing your communication structure will result in better processes and infrastructure almost regardless of the infrastructure.

Conway's Law states "any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure."[10]

DevOps as a first step

Agile methodology was an approach to software design that tried to pull all invested parties—QA, product management, developers, even documentation—into a more cohesive group. The idea was to clarify goals by having short iterations focused on specific tasks expressed as user goals (called stories). This broke down the traditional waterfall method of developing software, which was a kind of fire brigade of one team handing off to another team.

7 *Laycock, Rachel. "Continuous Delivery." Afternoon session. Red Hat Summit - DevNation 2016, 1 July 2016, San Francisco, California. https://www.youtube.com/watch?v=y87SUSOfgTY.*

8 *"Gartner Highlights Five Key Steps to Delivering an Agile I&O Culture." 20 Apr. 2015, www.gartner.com/newsroom/id/3032517.*

9 *DevNation Federal, June 8, 2017, Washington, DC, https://www.youtube.com/watch?v=tQ0o2qaUc6w&t=1s*

10 *Conway, Melvin E. (April 1968), "How do Committees Invent?", Datamation*

"Team designs are the first draft of your architecture."

– MICHAEL NYGARD, RELEASE IT!

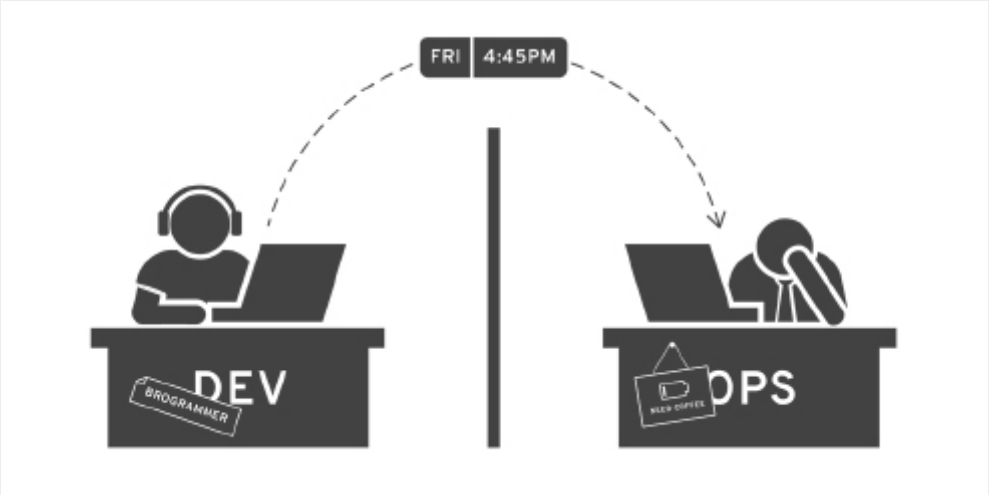

Figure 3. The wall of division, pre-DevOps

However, that still only addressed half of the actual life cycle of an application. Once something was developed, it got thrown over to operations to deploy and maintain (usually in a weekend maintenance window).

The problem is that operations does not always know what the application is supposed to do, which means they can deploy it inefficiently. Developers may have no clue about the real operating environment and may create an application that does not perform in the production environment. And then, to try to mitigate the risk of change, many organizations institute an onerous process of change management to try to explain and justify any change.

DevOps is a cultural shift that tries to break down the separation between developers, operations, and business stakeholders. The separations between those groups are real, but artificial. A team can include more than people who share a job function; DevOps tries to redefine the team to include everyone who shares in the life cycle of an application and open up communication between those groups.

Culture change goes even deeper than DevOps or agile or other methodologies. It is a commitment to actually putting everyone on the same team. If you change your communication patterns, you can change your outcomes.

Major changes can begin with very simple steps. Culture changes underpin all of the technological and process changes. If you're struggling to build a DevOps culture, try two things:

- Have your developers spend the weekend with operations, watching a production rollout and learning what they go through.
- Track how many steps or service tickets it takes for a developer to request a new virtual system.

Seeing how other teams are functioning, in situ, can be a powerful force to encourage teams to change their processes or to open up communication.

HIGHLIGHTS FROM PUPPET'S STATE OF DEVOPS REPORT

- 2,555 times faster lead times
- 200 times more deployments
- 24 times faster recovery from failure
- 3 times lower change failure rate
- 22% less time on rework

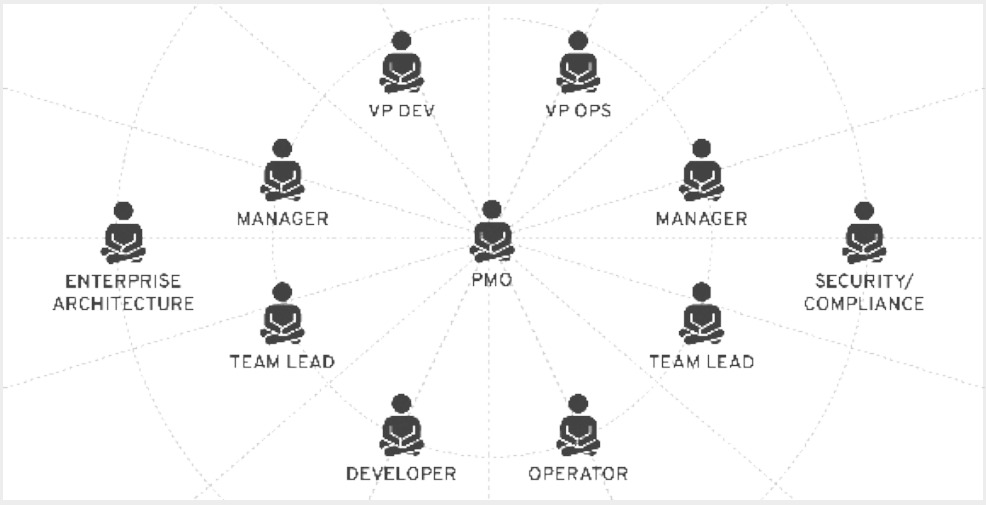

Figure 4. Full view of cross-team communication paths

Puppet's State of DevOps report shows how effective changing team structure (and communication) can be.[11] Its research reveals that DevOps teams experience:

- 2,555 times faster lead times.
- 200 times more deployments.
- 24 times faster recovery from failure.
- 3 times lower change failure rate.
- 22% less time on rework.

Speed is one of the key benefits of DevOps. Looking at the entire release process, all of those different teams need to be involved to push the application out the door, and depending on how clean the processes, infrastructure, and codebase are, that release process may take a lot of effort.

11 Kim, Gene, et al. "State of DevOps Report." Puppet, 2016, https://puppet.com/resources/whitepaper/2016-state-of-devops-report.

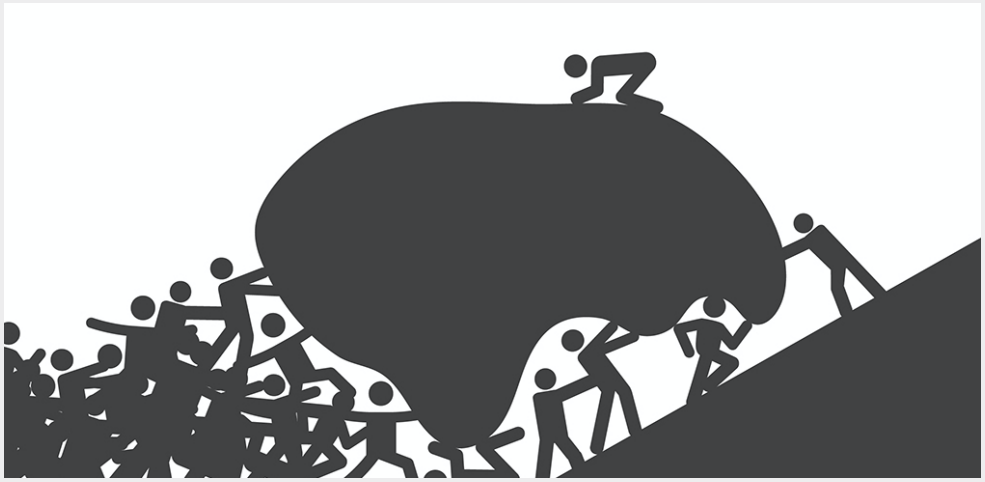

Figure 5. Team effort for a single application release

That pain was manageable when releases were relatively infrequent. Once software becomes a business driver, though, that entire release process has to happen many times per year. (In highly innovative companies like Amazon, that release process can happen hundreds of times per day.) In that case, the Herculean effort of multiple teams to move the application to production has to repeated again and again.

Figure 6. Effort for a single release, with more releases to come

DevOps changes the approach from an emergency, all-hands-on-deck push to a smoother, more sustainable iteration between group planning, development, testing, and deployment. That's why DevOps teams see astronomic growth in productivity—because the entire life cycle is repeatable.

Whether your ultimate application architecture is a more refined monolith or distributed microservices, changing your team structure and communication is a requirement. Communication has to be able to flow both before an application is planned and after it is deployed. A team per service can easily break into a silo per service, unless you've created and enabled a culture that supports open communication and feedback.[12]

DESIGNING AN APPLICATION ARCHITECTURE: LOOKING AT MICROSERVICES

A new application architecture is the final stage of digital evolution, but it is usually the most noticeable elephant in the room and one of the easiest to identify. It's worth looking at architecture, even if changing platforms and processes has to come first.

Design stamina, technical debt, and strategy

One huge impediment to change is that organizations look at their current applications and can't see a way to work around them. That's one reason many digital transformation initiatives consider a "rip and replace" approach; it sometimes seems easier to simply start over.

The problem, though, is that technical debt is a result of design. Taking out one application without a clear vision of what is going to replace it means that eventually the same impenetrable architecture will reemerge.

"Hope," explained Rachel Laycock, "is not a design method. You have to be very intentional."[13]

Whatever you are developing is your technical debt. For very new applications, teams frequently begin developing without a clear design, and that's not necessarily a bad thing. Martin Fowler's description of design stamina and technical debt points out that starting without a clear design frequently allows much faster initial innovation.[14] However, there comes a point where good design achieves a stable trajectory, and no design levels off.

---

12 Cotton, Ben. "From Monolith to Microservices." 3 Jan. 2017, https://www.nextplatform.com/2017/01/03/from-monolith-to-microservices/.

13 Laycock, Rachel. "Continuous Delivery." Afternoon session. Red Hat Summit - DevNation 2016, 1 July 2016, San Francisco, California. https://www.youtube.com/watch?v=y87SUSOfgTY

14 Fowler, Martin. "Design Stamina Hypothesis." 20 July 2007, https://martinfowler.com/bliki/DesignStaminaHypothesis.html.

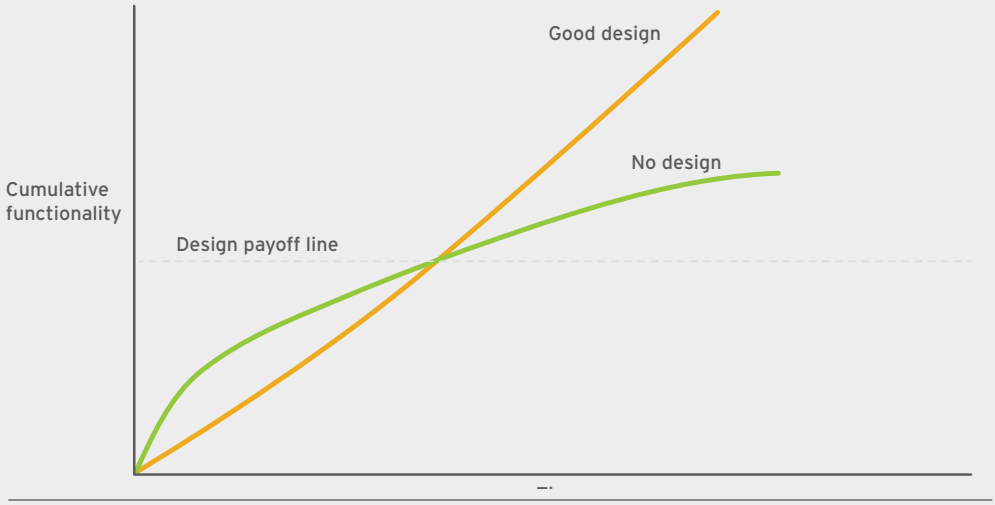

Figure 7. Conceptual graph of the design stamina theory

Before you start planning your architecture, you need to have a very clear understanding of your strategic priorities and goals. Rob Zuber wrote in Information Week:[15]

"If you don't have a clear understanding of your business, product, or where you fit in the stack, breaking things out prematurely into a bunch of services without knowing how they are going to be used is not going to be a fun outcome. … Right now is a great time to be thinking about architecture, but it's also important to do it in a way that is incremental and allows you to test ideas and validate them rather than trying to turn on a dime and build totally differently. That's going to fail and turn into a nine-month project that finishes with you telling your VP of engineering that, while you learned a lot, in the end the team decided not to ship it. That's not what you want to do. You want to be providing value and impact to your customers and business through planned and well-thought out processes."

In a recent IDC perspective, Stephen Elliot wrote that before ever defining an architecture, you must identify who the customer or end user is, what that customer values, what your desired outcomes are, and how you will measure success.[16]

In other words, you do not start with how you want to accomplish something or what architecture you want to use. You start with what your strategic goal is and then design the architecture that supports that. That puts the application first.

Microservices and monoliths, defined

There are three prominent application architectures today, based on the relationships between the services: monoliths (tightly coupled), microservices (decoupled), and (though falling out of favor) service-oriented architectures (loosely coupled).

15 Zuber, Rob. "Transitioning to Microservices: The Story of Two Monoliths ." InformationWeek, 25 May 2017, https://www.informationweek.com/devops/transitioning-to-microservices-the-story-of-two-monoliths-/a/d-id/1328972.

16 Elliot, Stephen. "Enabling DevOps with Modern Application Architecture." IDC Perspective, Dec. 2016.

INTENTIONAL PLANNING

- Who are your customers or users?
- What are they trying to do?
- What infrastructure do you have?
- What's the life cycle of that infrastructure?
- What services or functionality are required for a single workflow?
- What is the life cycle for that workflow?
- What is your deployment path, and how frequently does it need to be deployed?
- What business capabilities does this affect?

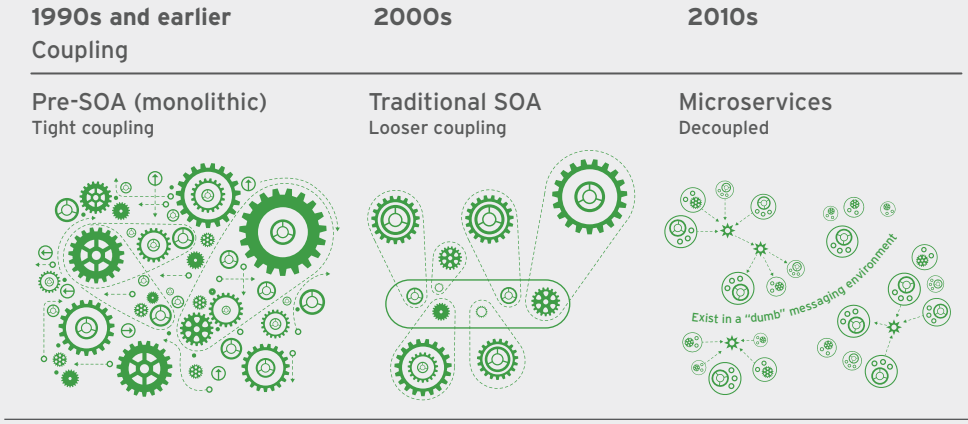

Figure 8. Coupling in monolith, service-oriented, and microservices architectures

In simple terms, a monolith is a single application stack that contains all functionality within that one application. This is tightly coupled, both in the interaction between the services and how they are developed and delivered. Updating or scaling a single aspect of the monolith application has implications for the entire application and its underlying infrastructure.

Dynamic scale and failover are potential issues with a monolith. Those tend to be addressed with simple scalability designs, such as horizontal scale (duplicating that function in a cluster) or vertical scale (mirroring instances and expanding hardware). A less frequently considered scalability issue, too, is that of the development and operations teams. If it takes a full 50 person team to release a monolithic application every 6-9 months, then it may be possible to improve the scale by having five smaller applications with five separate teams and ship individual updates every few weeks.

Monolithic architecture is probably the oldest application architecture, because of its initial simplicity and clearer relationships between services and interdependencies. This architecture is also more reflective of a limited, commodity-based IT infrastructure with more rigid development and release processes.

Because monoliths are an older style of architecture, they are frequently associated with legacy applications. In contrast, more modern architectures try to break out services by functionality or business capabilities to provide more flexibility. This is especially common with customer-facing interfaces—such as APIs, mobile apps, or web apps—which tend to be smaller and require more frequent updates to meet customer expectations.

One of the more current definitions of a distributed architecture is microservices. There are some similarities with other modular designs, like service-oriented architectures (SOA), but microservices move from a loose coupling between services to service independence. The definition of a single service is generally clear, and services can be added, upgraded, or removed from the larger architecture with ease. This has benefits for both dynamic scalability and fault tolerance: individual services

can be scaled as needed without requiring heavy infrastructure or can failover without impacting other services. As Ben Cotton wrote, "Microservice-based design is the ultimate manifestation of everything you learned about good application design."[17]

The fluidity of a microservice architecture means that it is closely associated with dynamic technologies like containers and clouds, which allow individual instances to be spun up and destroyed with ease, even programmatically.

The immediacy of distributed computing has direct benefits, both to the organization and to teams, who can see the impact of their work, including:

- Improved fault tolerance and minimized service disruptions.
- Simple protocols like JSON/REST and HTTP/OAuth for easier integration.
- Polyglot services, which allow for developer flexibility.
- Faster time to market for applications and functionality.
- Simplified data retrieval and sharing between services, without having to use heavier message buses or conversion.[18]

In Coding the Architecture, it states that a monolith architecture is contrasted with a microservices architecture when the runtime for the application itself is monolithic and static.[19] Many applications have numerous packages or modules that may be independent from one another in how they are built or deployed, but the application itself interacts as a single entity.

The underlying question is which architectural style is most appropriate. The knee-jerk reaction is to assume that new is always best, but the important thing is to step back and evaluate what fits your desired business outcomes. Engineers at Etsy and Netflix had an enjoyable Twitter fight over the necessity of microservices for continuous deployment and developer autonomy, with the engineers at Etsy pointing out that they had small, functionality-based, agile development teams and extremely rapid deployments (about 60 a day) while still running a monolithic application. They found a system which worked for them and their culture.

Architecture and technologies change and evolve. "Yesterday's best practice is tomorrow's antipattern," pointed out Laycock in her DevNation session address. "Microservices are a choice. They're an answer. They're not the solution; they're a potential solution."[20]

---

17 *Cotton, Ben. "From Monolith to Microservices." 3 Jan. 2017, https://www.nextplatform.com/2017/01/03/from-monolith-to-microservices/.*

18 ***Lambert, Natali**e. "Micro Services: Breaking down Software Monoliths." NetworkWorld, 22 Nov. 2017, https://www.networkworld.com/article/3143971/application-development/micro-services-breaking-down-software-monoliths.html.*

19 *Annett, Robert. "What Is a Monolith?" Coding the Architecture, 19 Nov. 2014, http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html.*

20 *Laycock, Rachel. "Continuous Delivery." Afternoon session. Red Hat Summit - DevNation 2016, 1 July 2016, San Francisco, California. https://www.youtube.com/watch?v=y87SUSOfgTY*

The question for an organization is not *can we replace our monolith with microservices?* The question is *what is our strategic goal and what do we need to do to get there?* Understanding the communication structure and culture, business capabilities, and needed workflows then influence how services are coupled, what their life cycles are, and, finally, the application architecture.

TABLE 1. COMPARISON OF MONOLITHIC AND MICROSERVICE ARCHITECTURES

	MONOLITHS	MICROSERVICES
Development	• Faster initial development • Difficult to change or add features	• Early design critical • Adding or changing services is easier
Application workflows	• Easy to fit application into workflows • Implement functionality (e.g., authentication or monitoring) in a single location	• More complicated to assign services within workflows • Interdependencies or requirements between services may not be clear
Training and maintenance	• Simple architecture • Rigid development requirements for environment and language	• Flexible architecture with more design complexity • Polyglot services with standardized APIs or messaging to connect
Scale	• Difficult to scale; dependent on hardware infrastructure • Scale entire application for single service spikes	• Easy to scale individual services without affecting overall architecture • Use software-defined infrastructure (containers, cloud) for dynamic responsiveness
Updates, failover, downtime	• All services are tightly coupled • Services must be updated together; versioning is coupled • Risk of system failure if there is an individual service failure	• Services are uncoupled • Services can be added or updated independently • Risk of failure is limited to a smaller number of services
Automation	• Automation is largely unnecessary	• Automation and orchestration is required

Fallacies of distributed computing

One engineer quipped that microservices turn every outage into a murder mystery.[21] That is a memorable summary of the biggest problem with distributed computing: dispersed complexity.

21 @HonestStatusPage. Twitter, 7 Oct. 2015, https://twitter.com/honest_update/status/651897353889259520?lang=en.

Increased costs and transaction costs

The required changes in infrastructure for true distributed computing can come at substantial capital costs. Also, there are indirect costs of retraining or acquiring new skills, changing team structures, and migrating systems. These costs can result in future savings in indicators like time to market and reduced down time (if the new architecture is effectively implemented), but those benefits are not immediate.

Increased complexity

Instead of a single (catastrophic) point of failure, as with a monolith, a microservices architecture has hundreds of different potential points of failure. Similar to a monolith, tracking down that failure can be difficult because it may not be obvious what the root cause is; however, microservices add extra complexity because the interdependencies between services are even less apparent.

Even the faster release cycles and more agile team structure introduce complexity. Effective communication is critical with microservices. Every software architecture is a matter of balancing inherent complexity. That complexity can either be hidden in the application itself (monoliths) or it can be pushed into the team communication structures (microservices).

Systems thinking and design

Designing an effective microservices architecture requires significant systems thinking. There has to be an understanding of interactions between services, business capabilities, and user experience. This systems thinking also has to extend to the communication structures and processes within the organizational culture. Gartner notes that without an understanding of the overall system, microservices architectures will perform a lot like the stereotypical monolith: "Failing to take a holistic approach that considers the software architecture, development infrastructure, and development processes will not provide optimal results, and you'll continue to suffer from many of the drawbacks of a monolithic software system."[22]

Resource constraints

The resource constraints with monoliths are obvious because of the problems with scale. Either the system has enough hardware capacity to handle the peak loads on the largest service, resulting in unused capacity, or it risks not having enough capacity to handle usage spikes.

With microservices, the architecture is flexible, and since each individual service is very small, it is easy to scale new services on very lightweight and temporary resources, such as containers or cloud instances.

Because the individual resource requirements for a given service are (relatively) small, it is possible to overlook or minimize the unique resource demands of the overall architecture. That leads to some assumptions:

- The network is always reliable.
- Latency is zero.
- Bandwidth is infinite.
- The network is secure.

---

22 Knoernschild, Kirk. "Refactor Monolithic Software to Maximize Architectural and Delivery Agility." Gartner Key insights, 18 May, 2017

- The infrastructure topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogenous.

There is another resource constraint, though it affects a niche use case for computing. Extremely high performance applications may have requirements that are too demanding for a microservices architecture, such as climate modeling or DNA mapping.

The fallacies of microservices are important to note only because no system is perfect. The intent should not be to find a perfect technology (or architecture) that fixes your problems. Rather, focusing on your culture and communication and refining your processes will create a mature, effective organization. From there, your organization will have the capacity to design a functional architecture that meets your specific objectives.

Rethinking a meaningful application

Much of the discussion around digital transformation focuses on the infrastructure and the culture, which makes sense given that they are foundational considerations. However, infrastructure and culture are the means; the end is to create an application which is useful to its users and relevant to the organization.

Meaningful applications possess certain characteristics:

- Responsive to users
- Reflective of the core business function or purpose
- Adaptive or reactive to dynamic changes in the environment
- Connected across environments
- Lightweight and flexible so that functionality can be added and maintained

When an application meets those characteristics, it is meaningful.

Both monolith and microservices architectures can reflect those characteristics. The architecture is a design choice. Even with a monolithic architecture, shifting your perspective on the different elements is critical to create a modern, nimble application. Etsy can run its continuous delivery, dozens of updates per day, high user loads, and even mobile applications because it has a majestic monolith—an intentionally architected application.

This intentionality must be reflected in either a monolithic or microservices architecture in several key areas, including:

- Integration.
- Data management and consistency.
- Messaging and service communications.

- Coherent process or workflow patterns.

In a gross generalization, traditional monolithic environments tended to approach these issues either as problems that needed to be fixed or as an opaque aspect of the application. Integration, for example, was sometimes viewed as an alternative to merging different applications or data sources; it was a bandage that pieced different parts of the environment together. Even things like process flows were viewed as a way of boosting productivity over manual intervention and automating workflows, but it was not always a core design decision.

With a modern application, aspects of the architecture, like integration and process flows, aren't secondary; they are central to how the application performs. This is very clear in microservices architectures (but it holds true for majestic monoliths as well).

Integration and messaging

With microservices, the tension is about how to couple those services in a way that maintains the autonomy of those services while still allowing free communication. That is an integration and a messaging issue. Integration defines how those separate services communicate with each other. This is a critical design choice. From an infrastructure perspective, microservices are sometimes broken down as "containers plus APIs," capturing the central need for coupling those services, with containers being the services and APIs being the connective tissue. (A similar approach would be "containers plus messaging"—any technology that provides a way to transmit data between services.)

This is different from even service-oriented architectures in that messaging and integration are not centralized in buses and data are not converted between services.

Process flows

A modern application is responsive to its users. This is most visible in mobile applications, which are consumer focused, simple, and immediate. When a customer initiates a transaction—such as checking a bank balance or browsing for an airline ticket—then a workflow has to be launched immediately. That workflow has to be adaptable to the next decision by the user, yet also conform to the protocols of the organization. Traditional business process management (BPM) was primarily a way to automate tasks for efficiency, but flipping that around in a modern application architecture, BPM becomes a major element in delivering functionality and contributing to the user experience.

Data

At some point, all data become stateful. Data must be stored and accessible by services within the architecture (whether they are distributed or monolithic), so there has to be an understanding of how data move between services and what kind of data are being generated.[23] Developer Christian Posta wrote that the hardest part about microservices is data management, trying to define the natural boundaries and transactional limits between services.[24] The tension here is between data consistency, accessibility, and autonomy. Defining the natural domains then informs the data models and storage structures.

---

23 Brown, Kyle. "Refactoring to Microservices, Part 2: What to Consider When Moving Your Data." IBM developerWorks, 4 May 2016, https://www.ibm.com/developerworks/cloud/library/cl-refactor-microservices-bluemix-trs-2/index.html.

24 Posta, Christian. "The Hardest Part About Microservices: Your Data." 14 July 2016, http://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/.

"People try to copy Netflix, but they can only copy what they see. They copy the results, not the process."[27]

– ADRIAN COCKCROFT, FORMER NETFLIX CHIEF CLOUD ARCHITECT

DRIVING FAST (RESPONSIBLY)

One of the primary objectives with digital transformation is to release applications faster. However, speed is just improvements in efficiency. To be transformative, speed has a purpose—it allows rapid innovation, new features, and an ability to test new ideas.

Ron Kohavi, a distinguished engineer and general manager of Microsoft's experimentation team for artificial intelligence, noted in an address to the University of Minnesota's Computer Science and Engineering Technology Open House in 2013 that less than a third of ideas improve the metrics that they were designed to improve.[25]

The way to evaluate a good idea is through thorough and effective testing—not just of the quality of code, but of the user experience and preference. This is only knowable through experience. As Kohavi puts it, "data trumps intuition."

This is the purpose of continuous delivery and advanced deployment techniques. CI/CD is the platform for rapid deployment; the deployment techniques are tools for experimentation and refinement.

Behind those two stages is culture change, which encourages innovation and supports failure and risk. Innovation is not a destination or a single point; it is a process that is fed with experimentation. Being willing to risk failure on the path to innovation requires a culture of humility.

Self-service, automation, and CI/CD

One of the initial restructures for digital transformation is moving to a DevOps culture, with small dynamic teams and cross-communication. The next stage is technology, providing an infrastructure that supports rapid development cycles.

There are two closely related technology stages with:

- Elastic, self-service infrastructures, which is the ability to request and receive an instance according to exact specifications almost instantly.
- Automation or orchestration, which is the ability to automatically create and manage multiple instances across an environment.

These are complementary technologies—you can't automate without an elastic environment, and managing potentially hundreds of instances is difficult without tools to make it consistent and repeatable.

Improving the technology at this stage of transformation can increase productivity tangibly. With one Red Hat customer, introducing a self-service catalog so that developers could rapidly request virtual systems lowered the time to get a requested system from five days to about 15 minutes and changed the process from manual to automatic.[26] That change frees up resources on the operations side and improves the productivity (and morale) of developers.

---

25 *"Online Controlled Experiments: Introduction, Insights, Scaling, and Humbling Statistics." SOBACO University of Minnesota, 18 Oct. 2013, https://sobaco.umn.edu/content/online-controlled-experiments-introduction-insights-scaling-and-humbling-statistics.*

26 *"Red Hat Virtualization Increases Efficiency and Cost-Effectiveness." Forrester Total Economic Impact Study, 26 Jan. 2017, www.redhat.com/en/resources/virtualization-tei-forrester-analyst-paper.*

27 *https://twitter.com/kelseyhightower/status/641886057391345664*

"Docker is best described as containers with an opinion because its set of tools encourage and build on the principles of an immutable infrastructure in which application configuration is fixed by the developer and rolled out to production with minimal additional configuration changes."[28]

–IDC

While the technologies are complementary, they are not prescriptive. Elastic infrastructures can mean cloud (public or private), virtual machines, or containers. Automation can mean a component within the infrastructure system (such as the Heat project for orchestration with OpenStack®) or it can be an external tool, such as Red Hat CloudForms or Kubernetes with Docker-based containers. There are different technological avenues, depending on your organization's skillsets and existing infrastructure.

One reason that containers (like Docker or Red Hat OpenShift) have been so associated with CI/CD is because they provide rigid and repeatable system environments, which means that there are fewer issues moving between entirely different organizational environments. (That's the dreaded "it worked on my laptop" defense when a code change moves from development to production.) Virtual machines or cloud instances can have variation between underlying operating system and package versions; containers must have identical operating system settings, and the selected container image is the same between deployments.

That consistency provides a good foundation for the first part of CI/CD—continuous integration. With continuous integration, development changes are constantly compiled and built with every check-in, so problems are apparent quicker. This is usually combined with an automated test suite to verify stability or functionality. This continual process of check-in - build - test maintains a higher quality of code.

Once the continuous integration path is running, then your organization can have continuous deployment, getting changes into production faster. This speed benefits both developers and operations. Developers and business leaders have the satisfaction of seeing new products go to market more quickly. Operations has the ability to roll out fixes and even critical common vulnerabilities and exposures (CVEs) in a much shorter time, enabling a more secure and performant system.

Continuous can mean slightly different things, depending on your development pace and business needs. With a majestic monolith (strong processes and technology with a more traditional application architecture), a release can happen every week with a single, monolithic update, with only the requirements for agile sprint processes working as a constraint.[29] With microservices, any service may be updated, with overlapping sprint cycles, so updates to the overall architecture can happen daily.

Advanced deployments and innovation

The stages leading to CI/CD are about speed and quality in delivering applications. But as Kohavi said, the majority of ideas do not accomplish what they were intended to accomplish. He illustrated this in his keynote address by providing a series of images of different designs of the Bing search engine, and asking the audience to guess, on instinct, which version users actually preferred. He gave four questions in sequence and, out of hundreds of attendees, only a single person remained standing.[30]

---

28 "The Emergence of Microservices as a New Architectural Approach to Building New Software Systems" in the INDUSTRY DEVELOPMENTS AND MODELS series by IDC. Author, Al Hilwa, June 2015.

29 Spazzoli, Raffaele. "The Fast-Moving Monolith: How We Sped-up Delivery from Every Three Months, to Every Week." Red Hat Developer's, 27 Oct. 2016, https://developers.redhat.com/blog/2016/10/27/the-fast-moving-monolith-how-we-sped-up-delivery-from-every-three-months-to-every-week/.

30 "Online Controlled Experiments: Introduction, Insights, Scaling, and Humbling Statistics." SOBACO University of Minnesota, 18 Oct. 2013, https://sobaco.umn.edu/content/online-controlled-experiments-introduction-insights-scaling-and-humbling-statistics.

His point was that data trumped intuition. The Bing design was finalized through extensive user testing, and they found that only about one-third of their ideas improved the user experience (and an equal amount actively damaged it).

Application infrastructure for experimentation

In 2006—nearly a decade before the term microservices would break into development consciousness—developer Neal Ford defined something called polyglot programming on his Meme Agora blog.[31] He identified a shift in the programming environment.

Very early in enterprise applications, applications were truly single language. As applications started shifting to server-client or web-based architectures, there were some languages that were specific to certain actions (like JavaScript for a web user interface or SQL for databases), but the core application was still written in a single, central language.

Ford's polyglot programming is the idea that there will no longer be a central programming language, but that within the overall application architecture, individual services or functions may be written in entirely different languages that best meet the needs of that particular service.

Polyglot programming is representative of one of the core needs for an agile development environment: developers have to be able to try something new and outside the core design of an application. This is as true for effective monolithic applications as for microservices architectures.

It is critical to consider the flexibility and options within the development environment when creating a platform for experimentation. An experimentation environment must support:

- Multiple languages.
- Multiple runtimes.
- Flexible deployment environments, such as cloud, physical, or hybrid.
- Flexible or changeable application architectures; this allows an application to adapt to environmental changes throughout its life cycle.
- Open standards or the ability to standardize iteratively.

Containers are a good example of this support, though it is possible to achieve the same outcome with other platforms. An individual container is inherently prescriptive about the libraries, languages, and versions within it. However, a container catalog can have hundreds or thousands of different images, supporting different languages and runtimes. This kind of platform allows developers to find the exact runtime and language that execute the service as desired, and it allows operations teams to effectively deploy those services.

Deployment patterns for innovation

Advanced deployment techniques bring structure and clarity to innovation. Mature deployment methodologies create an environment that allows true experimentation, feedback, and analysis. Better experimentation leads to better innovation.

These are common deployment patterns; any or all could be appropriate, depending on the nature of your application and your user environment.

---

31 Ford, Neal. "Polyglot Programming." Meme Agora, 5 Dec. 2006, http://memeagora.blogspot.com/2006/12/polyglot-programming.html.

"*Data trumps intuition.*"

RON KOHAVI, GENERAL MANAGER OF ARTIFICIAL INTELLIGENCE EXPERIMENTATION, MICROSOFT UNIVERSITY OF MINNESOTA, DEPT. OF COMPUTER SCIENCE AND ENGINEERING TECHNOLOGY OPEN HOUSE 2013 [23]

Blue-green environments

A blue-green environment is a way of mitigating the risk from rolling out changes. A new build passes through all of the environments in the CI/CD pipeline. For production, there are two identical environments (blue and green), but only one is active. The change is rolled out to the idle environment in production; once that environment is verified, then the router is switched and the traffic is moved to the updated environment.

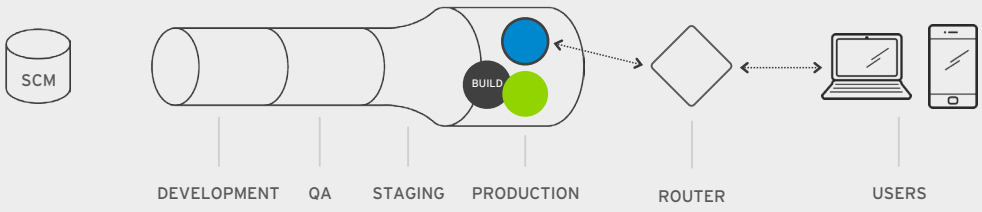

Figure 9. Blue-green deployment pipeline

Canary releases

A canary release is similar to a blue-green deployment, except the initial release only goes to a subset of users within the environment (the titular canaries in the coal mine). As feedback is gathered from the users, that subset can be increased incrementally until all users are eventually switched over.

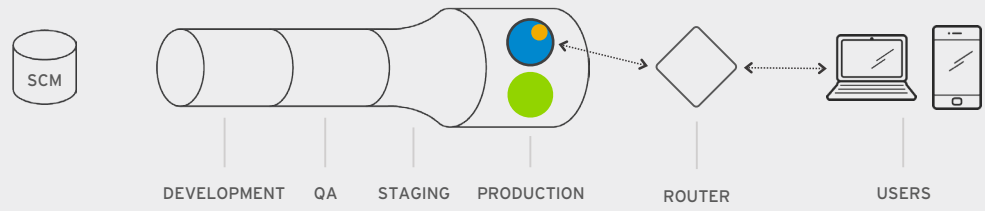

Figure 10. Canary deployment pipeline

This can be used as part of a testing technique to assess different functionality or designs for applications to small groups within an active production environment and with real traffic and usage patterns.

A/B testing

A/B testing presents users with two different designs and then evaluates which design performs better according to the desired metrics. This could be as explicit as having users rate their experiences or provide feedback, but it can also be done more subtly. For example, combining A/B testing with canary releases can compare two potential designs or even hidden functionality and then assess how users interact with the different designs, with the current environment as a baseline.

For example, in Figure 11, the mobile application seems identical to users, but the A/B environments are using different algorithms to test product recommendations.

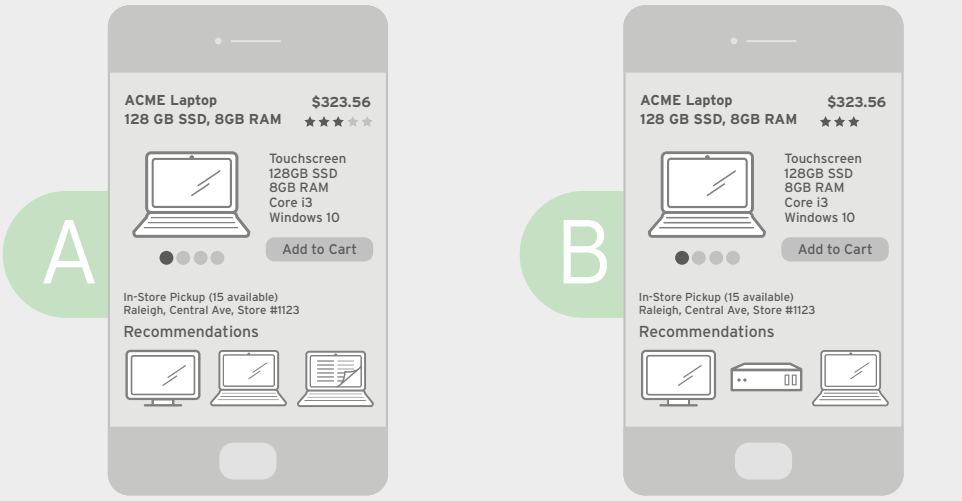

Figure 11. A/B interface testing

Once a given design is successful, then it can be released to a larger set of the environment, as with a canary release.

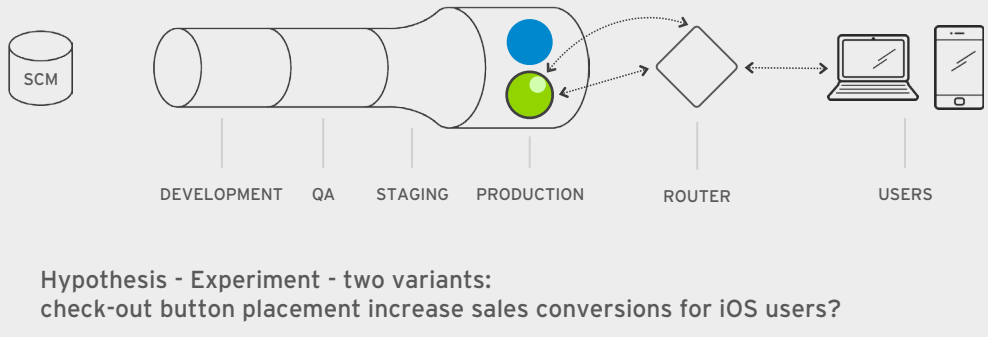

Figure 12. A/B testing release pipeline

When done effectively, this can turn a production environment into an experimentation environment, and it allows teams to be more innovative and more relevant in their designs.

This type of testing is the core of valuing data over intuition.

HOW TO TEACH AN ELEPHANT TO DANCE

Choose your stage

There are a lot of stages between a more traditional, waterfall environment and fully distributed microservices. Laycock discussed all software architecture being the result of the "tension between coupling and cohesion."[32] As you get started planning a digital transformation strategy, that tension is represented in the infrastructure and culture that you currently have.

Determine where your organization can realistically go. This doesn't mean "easily," since the goal of digital transformation is to significantly change the culture, processes, architecture, and technology. It means to understand what you are trying to achieve with that change and then clearly assess what it would take to move toward that goal. Ask yourself:

- What are your current team or group divisions?
- What are the communication patterns between those groups?
- Who is currently involved in planning cycles?
- Looking at functionality, how close is your current application architecture to your desired a pplication architecture?
- What is the level of risk or failure tolerance within your organization?
- How well understood are your material and information flows? (This is value mapping your organization.)
- How frequently do you need to be able to release an update to meet customer or operational needs?
- What new functionality is required by either business objectives or development needs?

Define your operating principles

Culture changes underpin all of the process, technology, or architecture changes that your organization will make for digital transformation.

While basic, creating a set of core principles that are backed by management and supported across teams can help reinforce digital transformation initiatives and unify teams.

32 Laycock, Rachel. "Continuous Delivery." Afternoon session. Red Hat Summit - DevNation 2016, 1 July 2016, San Francisco, California. https://www.youtube.com/watch?v=y87SUSOfgTY

"Strong application architecture, mature DevOps and agile processes, and a focused data management team all create an effective microservices environment. This environment can cut development lead time by 75%."

– GARTNER[33]

These principles can be simple, but it can be useful to be clear and explicit on the attitudes and behaviors that are most important for culture change, especially if some of these (like encouraging risk and experimentation) are contrary to the current culture. For example:

- Failure happens.
- Experimentation is good.
- Organization (people) are first.
- Practice continuous improvement.
- Commit to life-long learning.
- Always be accountable.
- Be transparent.

Figure 13. Smoother effort for a well-orchestrated release

33 Innovation Insights for Microservices, Anne Thomas and Aashish Gupta, January 27, 2017

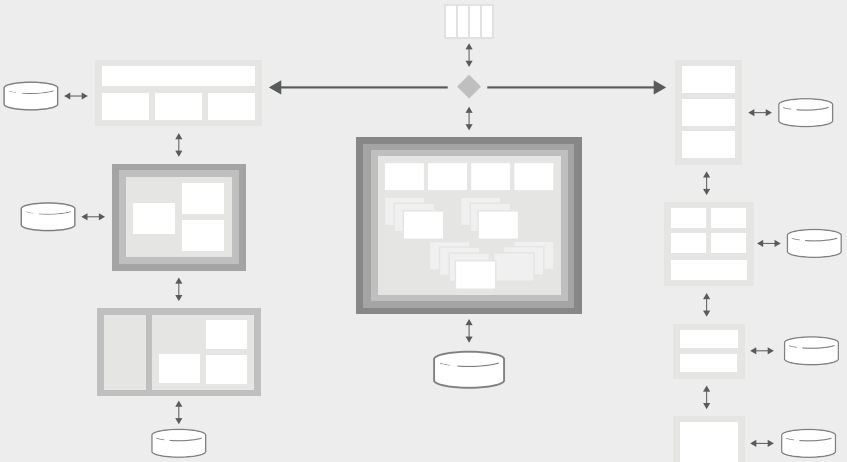

Figure 14. Breaking out individual services to "strangle" the monolith

Strangle your monolith

At some point, digital transformation will affect your current application and architecture. If you have an existing monolith application, there are two related ways to begin addressing that technical debt:

- Break out existing services as they require updates or replacement (strangling).
- Create new functionality in separate, independent services (starving).

This does not require a full commitment to microservices. Key Bank, a Red Hat customer, needed to reduce release times from quarterly to weekly, and it accomplished this while still having a monolithic application.[34] The core application logic may remain in a monolith, but there can be logical domains where services can be separated, most commonly for the user interfaces. For example, creating a separate layer for services like an API system, mobile front end, and other user interfaces allow customer- or external-facing services to have faster iterations or more separate life cycles than the core application, which allows more innovation with less business risk.

Gartner recommends this incremental approach to distributed architectures because it is both "iterative and focused on areas of value."[35]

34 Spazzoli, Raffaele. "The Fast-Moving Monolith: How We Sped-up Delivery from Every Three Months, to Every Week." Red Hat Developers, 27 Oct. 2016, developers.redhat.com/blog/2016/10/27/the-fast-moving-monolith-how-we-sped-up-delivery-from-every-three-months-to-every-week/.

35 Olliffe, Gary. "How to Design Microservices for Agile Architecture." Gartner Key Insights, 30 Jan, 2017.

Whatever the final stage of digital evolution for your organization, there are three basic areas that should be covered in your approach:

- Agility in architectural design.
- Experimentation.
- Automation.

Design architectures for future agility

Whether your focus is on streamlining processes to improve your monolith or on creating microservices, your architectural foundation has to be agile. Frequently, agility means hybrid. Gartner recommends starting even new projects as monoliths and breaking out microservices as they mature.[36] Gartner states, "Your initial reaction to this may be 'won't that be a waste of development effort?' In short, our research suggests the opposite: The monolith-first approach will reduce risk, improve initial productivity, and ensure you decouple and decompose your application into the right set of microservices."[37]

Remembering the design stamina theory, create a development process that emphasizes clarity and simplicity. The code should be easy to understand. The functionality and purpose should be clear. As the application matures, then it can evolve into more distributed architectures. Having good development and deployment processes will keep that pathway agile.

Set aside both time and budget for experimentation

There has to be space in the budget for experimenting with new technologies and application features. For example, IDC recommends setting aside 2% of your IT budget just for experimenting with container technologies.[38]

Gartner states that technology isn't a goal in itself, so attempting a change to "deploy in the cloud" or "do microservices" will fail because of the lack of clarity in what those changes are supposed to accomplish.[39]

However, digital strategic goals are very frequently supported by technology changes. Reducing time to market may require moving to containers (for example), and JavaTM EE applications can run in containerized platforms like Red Hat JBoss® Enterprise Application Platform (EAP).

Set aside resources that allow your developers and operations groups to identify useful technologies and to develop their skills to support whatever infrastructure is ultimately deployed.

Automate everything

Automation has (among others) two very clear benefits: improved efficiency by removing manual steps and inherent consistency and reproducibility. Automating every step for development (to start) and deployment (as processes mature) will give your teams feedback loops on changes at every step and as stakeholders shift from development to operations to customers. This approach improves overall code quality.

---

36 Ibid.

37 Ibid.

38 Elliot, Stephanie, et al. "IDC TechBrief: Containers." IDC. Jan. 2017.

39 Knoernschild, Kirk. "Refactor Monolithic Software to Maximize Architectural and Delivery Agility." Gartner Key insights, 18 May, 2017.

As a first step, set a baseline for the current status of your organization to feed into your automation strategy. Steps include:

- Defining relevant metrics.
- Visualizing or diagramming your current workflows.
- Identifying key participants at different steps.

CONCLUSION

Over time, enterprise applications tend to devolve into the stereotypical monolith—opaque, cumbersome to update, and slow to incorporate new functionality. Yet, these enterprise applications also provide the core business and revenue-generating operations. This is the elephant in the room.

That elephant can be trained into something nimble and transformative, as long as there is a clear vision of what that final state should be. This is digital transformation as an evolutionary process. There is no one ideal outcome; each evolutionary path reflects the unique purpose and personality of the organization itself.

Assess each stage of digital evolution: DevOps, self-service or elastic environments, automation, CI/CD pipelines, advanced deployments, and microservices. Build your digital transformation strategy around the level of evolution that best matches your business needs.

Focus on building your culture, and balance technology changes with corresponding process changes so that your technology is fully supported by your teams.

As your processes mature, begin evaluating your application and architecture. As necessary, isolate or develop independent services, and create an agile architecture that can be adapted as business priorities change or emerge.

Lastly, foster an ability to innovate. This means having some tolerance for risk and failure (within the limits of your business objectives and customer needs). It requires the discipline to set aside resources in time, money, and infrastructure. Experimentation is at the root of innovation, and it sets up a better chance for digital transformation success. It also recaptures some of the initial joy that drew so many developers and operations people into technology in the first place—the ability to create and see that creation.

ABOUT RED HAT

Red Hat is the world's leading provider of open source software solutions, using a community-powered approach to provide reliable and high-performing cloud, Linux, middleware, storage, and virtualization technologies. Red Hat also offers award-winning support, training, and consulting services. As a connective hub in a global network of enterprises, partners, and open source communities, Red Hat helps create relevant, innovative technologies that liberate resources for growth and prepare customers for the future of IT.

NORTH AMERICA
1 888 REDHAT1

EUROPE, MIDDLE EAST, AND AFRICA
00800 7334 2835
europe@redhat.com

ASIA PACIFIC
+65 6490 4200
apac@redhat.com

LATIN AMERICA
+54 11 4329 7300
info-latam@redhat.com

Copyright © 2017 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Shadowman logo, and JBoss are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries. Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries. The OpenStack word mark and the Square O Design, together or apart, are trademarks or registered trademarks of OpenStack Foundation in the United States and other countries, and are used with the OpenStack Foundation's permission. Red Hat, Inc. is not affiliated with, endorsed by, or sponsored by the OpenStack Foundation or the OpenStack community.

facebook.com/redhatinc
@redhatnews
linkedin.com/company/red-hat

redhat.com
#8980_0917